Adding expert knowledge and exploration in Monte-Carlo Tree Search

Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, Olivier Teytaud

Adding expert knowledge and exploration in Monte-Carlo Tree Search

Guillaume Chaslot[1], Christophe Fiter[2], Jean-Baptiste Hoock[2], Arpad Rimmel[2], Olivier Teytaud[2]

[1] Games and AI Group, MICC, Faculty of Humanities and Sciences, Universiteit Maastricht, Maastricht, The Netherlands
[2] TAO (Inria), LRI, UMR 8623 (CNRS - Univ. Paris-Sud), bat 490 Univ. Paris-Sud 91405 Orsay, France, teytaud@lri.fr

Abstract. We present a new exploration term, more efficient than classical UCT-like exploration terms and combining efficiently expert rules, patterns extracted from datasets, All-Moves-As-First values and classical online values. As this improved bandit formula does not solve several important situations (semeais, nakade) in computer Go, we present three other important improvements which are central in the recent progress of our program MoGo:

- We show an expert-based improvement of Monte-Carlo simulations for *nakade* situations; we also emphasize some limitations of this modification.
- We show a technique which preserves diversity in the Monte-Carlo simulation, which greatly improves the results in 19x19.
- Whereas the UCB-based exploration term is not efficient in MoGo, we show a new exploration term which is highly efficient in MoGo.

MoGo recently won a game with handicap 7 against a 9Dan Pro player, Zhou JunXun, winner of the LG Cup 2007, and a game with handicap 6 against a 1Dan pro player, Li-Chen Chien. [1]

1 Introduction

Monte-Carlo Tree Search (MCTS [5, 7, 11]) is a recent tool for difficult planning tasks. Impressive results have already been produced in the case of the game of Go [7, 10].

MCTS consists in building a tree, in which nodes are situations of the considered environnement and branches are the actions that can be taken by the agent. The main point in MCTS is that the tree is highly unbalanced, with a strong bias in favor of important parts of the tree. The focus is on the parts of the tree in which the expected gain is the highest. For estimating which situation should be further analyzed, several algorithms have been proposed: UCT[11] (Upper Confidence Trees), focuses on the proportion of winning simulation plus an uncertainty measure; AMAF [4, 1, 10] (All Moves As First, also termed RAVE

[1] A preliminary version of this work was presented at the EWRL workshop, without proceedings.

for Rapid Action-Value Estimates in the MCTS context), focuses on a compromise between UCT and heuristic information extracted from the simulations; BAST[6] (Bandit Algorithm for Search in Tree), uses UCB-like bounds modified through the overall number of nodes in the tree. Other related algorithms have been proposed as in [5], essentially using a decreasing impact of a heuristic (pattern-dependent) bias as the number of simulations increases. In all these cases, the idea is to bias random simulations thanks to statistics.

In the context of the game of Go[2], the nodes are equipped with one Go board configuration, and with statistics, typically the number of won and lost games in the simulations started from this node (the RAVE statistics requires some more statistics). MCTS uses these statistics in order to iteratively expand the tree in the regions where the expected reward is maximum. After each simulation from the current position (the root) until the end of the game, the win and loss statistics are updated in every node concerned by the simulation, and a new node corresponding to the first new situation of the simulation is created. The algorithm is therefore as follows:

Initialize the tree T to only one node, equipped with the current situation.
while There is time left **do**
  Simulate one game g from the root of the tree to a final position, choosing moves as follows:
  Bandit part: for a situation in T, choose the move with maximal score.
  MC part: For a situation out of T, choose the move thanks to Alg. 1.
  Update win/loss statistics in all situations of T crossed by g.
  Add in T the first situation of g which is not yet in T.
end while
Return the move simulated most often from the root of T.

The reader is referred to [5, 7, 11, 10, 9] for a detailed presentation of MCTS techniques and various scores. We will propose our current bandit formula in section 2.

The function used for taking decisions out of the tree (i.e. the so-called Monte-Carlo part, MC) is defined in Algorithm 1. An atari occurs when a string (a group of stones) can be captured in one move. Some Go knowledge has been added in this part in the form of 3×3 expert designed patterns in order to play more meaningfull games.

Unfortunately, some bottlenecks appear in MCTS. In spite of many improvements in the bandit formula, there are still situations which are poorly handled by MCTS. MCTS uses a bandit formula for moves early in the tree, but can't figure out long term effects which involve the behavior of the simulations far from the root. The situations which are to be clarified at the very end should therefore be included in the Monte-Carlo part and not in the bandit.

We therefore propose three improvements in the MC part:

– Diversity preservation as explained in section 3.1;

[2] Definitions of the different Go terms used in this article can be found on the web site `http://senseis.xmp.net/`.

Algorithm 1 Algorithm for choosing a move in MC simulations, for the game of Go.

```
if the last move is an atari, then
  Save the stones which are in atari if possible (this is checked by liberty count).
else
  if there is an empty location among the 8 locations around the last move which
  matches a pattern then
    Sequential move: play randomly uniformly in one of these locations.
  else
    if there is a move which captures stones then
      Capture move: capture stones.
    else
      if there is a legal move then
        Legal move:Play randomly a legal move
      else
        Return pass.
      end if
    end if
  end if
end if
```

– Nakade refinements as explained in section 3.2;
– Elements around the Semeai, as explained in section 3.3.

2 Combining offline, transient, online learnings and expert knowledge, with an exploration term

In this section we present how we combine online learning (bandit module), transient learning (RAVE values), expert knowledge (detailed below) and offline pattern-information. RAVE values are presented in [10]. We point out that this combination is far from being straightforward: due to the subtle equilibrium between online learning (i.e. naive success rates of moves) transient learning (RAVE values) and offline values, the first experiments were highly negative, and become clearly conclusive only after careful tuning of parameters[3].

The score for a decision d (i.e. a legal move) is as follows:

$$score(d) = \alpha \underbrace{\hat{p}(d)}_{Online} + \beta \underbrace{\widehat{\widehat{p}}(move)}_{Transient} + (\gamma + \frac{C}{\log(2+n(d))}) \underbrace{H(d)}_{Offline} \quad (1)$$

where the coefficients α, β, γ and C are empirically tuned coefficients depending on $n(d)$ (number of simulations of the decision d) and n (number of simulations of the current board) as follows:

[3] We used both manual tuning and cross-entropy methods. Parallelization was highly helpful for this.

$$\beta = \#\{rave\ sims\}/(\#\{rave\ sims\} + \#\{sims\} + c_1\#\{sims\}\#\{rave\ sims\}) \quad (2)$$

$$\gamma = c_2/\#\{rave\ sims\} \quad (3)$$

$$\alpha = 1 - \beta - \gamma \quad (4)$$

where $\#\{rave\ sims\}$ is the number of Rave-simulations, $\#\{sims\}$ is the number of simulations, C, c_1 and c_2 are empirically tuned. For the sake of completeness, we precise that C, c_1 and c_2 depend on the board size, and are not the same in the root of the tree during the beginning of the thinking time, in the root of the tree during the end of the thinking time, and in other nodes. Also, this formula is computed most often with an approximated (faster) formula, and sometimes with the complete formula - it was empirically found that the constants should not be the same in both cases. All these local engineering improvements make the formula quite unclear and the take-home message is mainly that MoGo has good results with $\alpha+\beta+\gamma = 1$, $\gamma \simeq c_2/\#\{rave\ sims\}$ and with the logarithmic formula $C/\log(2 + n(d)$ for progressive unpruning. These rules imply that:

- initially, the most important part is the offline learning;
- later, the most important part is the transient learning (RAVE values);
- eventually, only the "real" statistics matter.

$H(d)$ is the sum of two terms: patterns, as in [3, 5, 8], and rules detailed below:

- capture moves (in particular, string contiguous to a new string in atari), extension (in particular out of a ladder), avoid self-atari, atari (in particular when there is a ko), distance to border (optimum distance = 3 in 19x19 Go), short distance to previous moves, short distance to the move before the previous move; also, locations which have probability nearly 1/3 of being of one's color at the end of the game are preferred.

The following rules are used in our implementation in 19x19, and improve the results:

- Territory line (i.e. line number 3), Line of death (i.e. first line), Peep-connect (ie. connect two strings when the opponent threatens to cut), Hane (a move which "reaches around" one or more of the opponent's stones), Threat, Connect, Wall, Bad Kogeima(same pattern as a knight's move in chess), Empty triangle (three stones making a triangle without any surrounding opponent's stone).

They are used both (i) as an initial number of RAVE simulations (ii) as an additive term in H. The additive term (ii) is proportional to the number of AMAF-simulations.

These shapes are illustrated on Figure 1. With a naive hand tuning of parameters, only for the simulations added in the AMAF statistics, they provide 63.9±0.5 % of winning rate against the version without these improvements. We

are optimistic on the fact that tuning the parameters will strongly improve the results. Moreover, since the early developments of MoGo, some "cut" bonuses are included (i.e., advantages for playing at locations which match "cut" patterns, i.e. patterns for which a location prevents the opponent from connecting two groups).

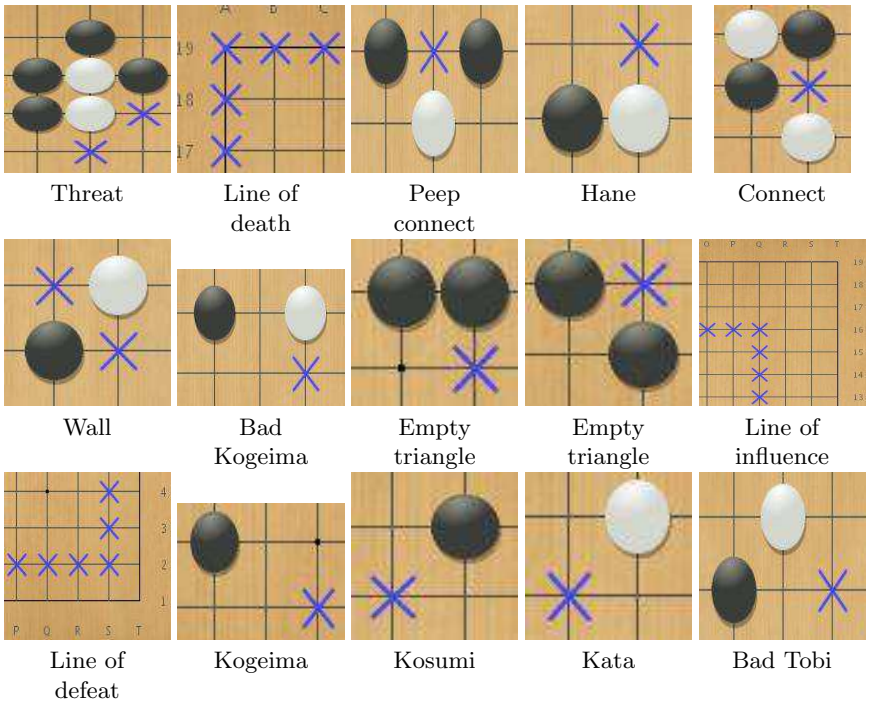


Fig. 1. We here present shapes for which exact matches are required for applying the bonus/malus. In all cases, the shapes are presented for the black player: the feature applies for a black move at one of the crosses. The reverse pattern of course applies for white. Threat is not an exact shape to be matched but just an example: in general, black has a bonus for simulating one of the liberties of an enemy string with exactly two liberties, i.e. to generate an atari.

Following [3], we built a model for evaluating the probability that a move is played, conditionally to the fact that it matches some pattern. When a node is created, the pattern matching is called, and the probability it proposes is used as explained later (Eq. 1). The pattern matching is computationnally expensive and we had to tune the parameters in order to have positive results.

The following parameters had to be modified, when this model was included in H:

– time scales for the convergence of the weight of online statistics to 1 (see Eq. 1) are increased;
– the number of simulations of a move at a given node before the subsequent nodes is created is increased (because the computational cost of a creation is higher).
– the optimal coefficients of expert rules are modified;
– importantly, the results were greatly improved by adding the constant C (see Eq. 1). This is the last line of table 2.

Results are presented in figure 2.

Tested version	Against	Conditions of games	Success rate
MoGo + patterns	MoGo without patterns	3000 sims/move	56 % ± 1%
MoGo + patterns	MoGo without patterns	2s/move	50.9 % ± 1.5 %
MoGo + patterns + tuning of coefficients	MoGo + patterns	1s/move	55.2 % ± 0.8 %
MoGo + patterns + tuning of coefficients + adding and tuning C	MoGo + patterns + tuning of coefficients	1s/move	61.72 % ± 3.1 %

Fig. 2. Effect of adding patterns extracted from professional games in MoGo. The first tuning of parameters is the tuning of α, β and γ as functions of $n(d)$ (see Eq. 1) and of coefficients of expert rules. A second tuning consists in tuning constant C in Eq. 1.

3 Improving Monte-Carlo (MC) simulations

There exists no easy criterion to evaluate the effect of a modification of the Monte-Carlo simulator on the global MCTS algorithm in the case of Go or more generally two players games. Many people have tried to improve the MC engine by increasing its level (the strength of the Monte-Carlo simulator as a stand-alone player), but it is shown clearly in [13, 10] that this is not the good criterion: a MC engine MC_1 which plays significantly better than another MC_2 can lead to very poor results as a module in MCTS, whenever the computational cost is the same. Some MC engines have been learnt on datasets [8], but the results are strongly improved by changing the constants manually. In that sense, designing and calibrating a MC engine remains an open challenge: one has to intensively experiment a modification in order to validate it.

Various shapes are defined in [2, 13, 12]. [13] uses patterns and expertise as explained in Algorithm 1. We present below two new improvements, both of them centered on an increased diversity when the computational power increases; in both cases, the improvement is negative or negligible for small computational power and becomes highly significant when the computational power increases.

3.1 Fill the board: random perturbations of the Monte-Carlo simulations

The principle of this modification is to play first on locations of the board where there is large empty space. The idea is to increase the number of locations at which Monte-Carlo simulations can find pattern-matching in order to diversify the Monte-Carlo simulations.

As trying every position on the board would take too much time, the following procedure is used instead. A location on the board is chosen randomly; if the 8 surrounding positions are empty, the move is played, else the following $N - 1$ positions on the board are tested; N is a parameter of the algorithm. This modification introduces more diversity in the simulations: this is due to the fact that the Monte-Carlo player uses a lot of patterns. When patterns match, one of them is played. So the simulations have only a few ways of playing when only a small number of patterns match; in particular at the beginning of the game, when there are only a few stones on the goban. As this modification is played before the patterns, it leads to more diversified simulations (Figure 3 (left)). The detailed algorithm is presented in Algorithm 2, experiments in Figure 3 (right).

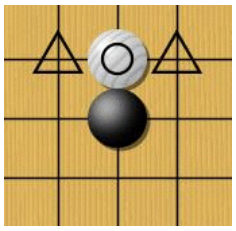
Algorithm 2 Algorithm for choosing a move in MC simulations, including the "fill board" improvement. We experimented also a constraint of 4, 12 and 22 empty locations instead of 8, but results were disappointing.

if the last move is an atari, **then**
  Save the stones which are in atari if possible.
else
  "Fill board" part.
  for $i \in \{1, 2, 3, 4, \ldots, N\}$ **do**
    Randomly draw a location x on the goban.
    IF x is an empty location and the eight locations around x are empty, play x (exit).
  end for
  End of "fill board" part.
  Sequential move, if any (see above).
  Capture move, if any (see above).
  Random legal move, if any (see above).
end if

3.2 The "Nakade" problem

A known weakness of MoGo, as well as many MCTS programs, is that *nakade* is not correctly handled. We will use the term *nakade* to design a situation in which a surrounded group has a single large internal, enclosed space in which the player won't be able to establish two eyes if the opponent plays correctly.

The group is therefore dead, but the baseline Monte-Carlo simulator (Algorithm 1) sometimes estimates that it lives with a high probability, i.e. the MC



9x9 board		19x19 board	
Nb of playouts per move or time/move	Success rate	Nb of playouts per move or time /move	Success rate
10 000	52.9 % ± 0.5%	10000	49.3 ± 1.2 %
5s/move, 8 cores	54.3 % ± 1.2 %	5s/move, 8 cores	77.0 % ± 3.3 %
100 000	55.2 % ± 1.4 %	100 000	73.7 % ± 2.9%
200 000	55.0 % ± 1.1 %	200 000	78.4 % ± 2.9 %

Fig. 3. Left: diversity loss when the "fillboard" option was not applied: the white stone is the last move, and the black player, starting a Monte-Carlo simulation, can only play at one of the locations marked by triangles. Right: results associated to the "fillboard" modification. As the modification leads to a computational overhead, results are better for a fixed number of simulations per move; however, the improvement is clearly significant. The computational overhead is reduced when a multi-core machine is used: the concurrency for memory access is reduced when more expensive simulations are used, and therefore the difference between expensive and cheap simulations decays as the number of cores increases. This element also shows the easier parallelization of heavier playouts.

simulation does not necessarily lead to the death of this group. Therefore, the tree will not grow in the direction of moves preventing difficult situations with *nakade* — MoGo just considers that this is not a dangerous situation.

This will lead to a false estimate of the probability of winning. As a consequence, the MC part (i.e. the module choosing moves for situations which are not in the tree) must be modified so that the winning probability reflects the effect of a *nakade* .

Interestingly, as most MC tools have the same weakness, and also as MoGo is mainly developed by self-play, the weakness concerning the *nakade* almost never appeared before humans found the weakness (see post from D. Fotland called "UCT and solving life and death" on the computer-Go mailing list). It would be theoretically possible to encode in MC simulations a large set of known *nakade* behaviors, but this approach has two weaknesses: (i) it is expensive and MC simulations must be very fast (ii) abruptly changing the MC engine very often leads to unexpected disappointing effects. Therefore we designed the following modification: if a contiguous set of exactly 3 free locations is surrounded by stones from the opponent, then we play at the center (the vital point) of this "hole". The new algorithm is presented in Algorithm 3.

We validate the approach with two different experiments: (i) known positions in which old MoGo does not choose the right move (Figure 4) (ii) games confronting the new MoGo vs the old MoGo (Table 1).

We also show that our modification is not sufficient for all cases: in the game presented in Fig. 4 (e), MoGo lost with a poor evaluation of a nakade situation, which is not covered by our modification.

Algorithm 3 New MC simulator, reducing the *nakade* problem.

if the last move is an atari, **then**
  Save the stones which are in atari if possible.
else
  Beginning of the *nakade* modification
  for x in one of the 4 empty locations around the last move **do**
    if x is in a hole of 3 contiguous locations surrounded by enemy stones or the sides of the goban **then**
      Play the center of this hole (exit).
    end if
  end for
  End of the *nakade* modification
  "Fill board" part (see above).
  Sequential move, if any (see above).
  Capture move, if any (see above).
  Random legal move, if any (see above).
end if

Number of simulations per move	Success rate	Number of simulations per move	Success rate
9x9 board		19x19 board	
10000	52.8 % ± 0.5%		
100000	55.6 % ± 0.6 %	100 000	53.2 % ± 1.1%
300000	56.2 % ± 0.9 %		
5s/move, 8 cores	55.8 % ± 1.4 %		
15s/move, 8 cores	60.5 % ± 1.9 %		
45s/move, 8 cores	66.2 % ± 1.4 %		

Table 1. Experimental validation of the nakade modification: modified MoGo versus baseline MoGo. Seemingly, the higher the number of simulations (which is directly related to the level), the higher the impact.

3.3 Approach moves

Correctly handling life and death situation is a key point in improving the MC engine . Reducing the probability of simulations in which a group which should clearly live dies (or vice versa) improves the overall performance of the algorithm. For example, in Fig. 5, black should play in B before playing in A for killing A. This is an approach move. We implemented this as presented in algorithm 4. This modification provides a success rate of

- 52.68 % (± 0.33 %) in 9x9 with 20 000 simulations per move;
- 54.69 % (± 2.27%) in 19x19 with 50 000 simulations per move.

We can see on Fig. 5 that some semeai situations are handled by this modification: MoGo now clearly sees that black, playing first, can kill on Fig. 5. Unfortunately, this does not solve more complicated semeais as e.g. Fig. 5 (e).

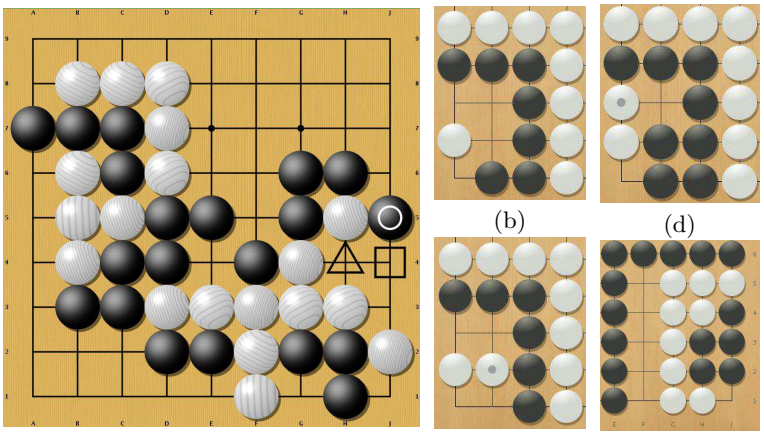


Fig. 4. In Figure (a) (a real game played and lost by MoGo), MoGo (white) without specific modification for the *nakade* chooses H4; black plays J4 and the group F1 is dead (MoGo looses). The right move is J4; this move is chosen by MoGo after the modification presented in this section. Examples (b), (c) and (d) are other similar examples in which MoGo (as black) evaluates the situation poorly and doesn't realize that his group is dead. The modification solves the problem. (e) An example of more complicated *nakade* , which is not solved by MoGo - we have no generic tool for solving the *nakade* .

4 Conclusion

First, as well as for humans, all time scales of learning are important: offline knowledge (strategic rules and patterns) as in [7, 5]; online information (i.e. analysis of a sequence by mental simulations) [10]; transient information (extrapolation as a guide for exploration).

Second, reducing diversity has been a good idea in Monte-Carlo; [13] has shown that introducing several patterns and rule greatly improve the efficiency of Monte-Carlo Tree-Search. However, plenty of experiments around increasing the level of the Monte-Carlo simulator as a stand-alone player have given negative results - diversity and playing strength are too conflicting objectives. It is even not clear for us that the goal is a compromise between these two criteria. We can only clearly claim that increasing the diversity becomes more and more important as the computational power increases, as shown in section 3.

Approach moves are an important feature. It makes MoGo more reasonnable in some difficult situations in corners. We believe that strong improvements can arise as generalizations of this idea, for solving the important semeai case.

Importantly, whereas exploration by a UCT term $+\sqrt{\log(\text{nbSims of father nodes})/\text{nbSims of child node})}$ as in UCB is important when scores are naive empirical success rates, the optimal constant in the exploration term becomes 0 when learning is improved (at least in MoGo,

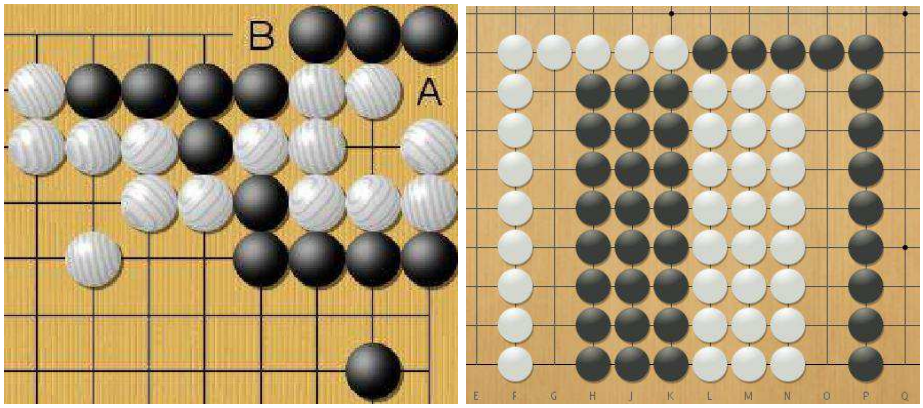


Fig. 5. Left: Example of situation which is poorly estimated without approach moves. Black should play B before playing A for killing the white group and live. Right: situation which is not handled by the "approach moves" modification.

and the constant is very small in several UCT-like programs also). In MoGo, the constant in front of the exploration term was not null before the introduction of RAVE values in [10]; it is now 0. Another term has provided an important improvement as an exploration term: the constant C in Eq. 1.

Acknowledgements. We thank Bruno Bouzy, Rémi Munos, Yizao Wang, Rémi Coulom, Tristan Cazenave, Jean-Yves Audibert, David Silver, Martin Mueller, KGS, Cgos, and the computer-go mailing list for plenty of interesting discussions. Many thanks to the french federation of Go and to Recitsproque for having organized an official game against a high level human; many thanks also to the several players from the French and Taiwanese Federations of Go who accepted to play and discuss test games against MoGo.

References

1. B. Bouzy and B. Helmstetter. Monte-Carlo Go developments. In Ernst A. Heinz, H. Jaap van den Herik, and Hiroyuki Iida, editors, *10th Advances in Computer Games*, pages 159–174, 2003.
2. Bruno Bouzy. Associating domain-dependent knowledge and Monte-Carlo approaches within a go program. In K. Chen, editor, *Information Sciences, Heuristic Search and Computer Game Playing IV*, volume 175, pages 247–257, 2005.
3. Bruno Bouzy and Guillaume Chaslot. Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. In *G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181, 2005.
4. B. Bruegmann. Monte-Carlo Go. *Unpublished*, 1993.

Algorithm 4 New MC simulator, implementing approach moves. Random is a random variable uniform on $[0, 1]$.

if the last move is an atari, **then**
  Save the stones which are in atari if possible.
else
  ***Nakade* modification (see above).**
  "Fill board" part (see above).
  if there is an empty location among the 8 locations around the last move which matches a pattern **then**
    Randomly and uniformly select one of these locations.
    if this move is a self-atari and can be replaced by a a connection with another group *and* random < 0.5 **then**
      Play this connection (exit).
    else
      Play the select location (exit).
    end if
  else
    Capture move, if any (see above).
    Random legal move, if any (see above).
  end if
end if

5. G.M.J.B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.
6. Pierre-Arnaud Coquelin and Rmi Munos. Bandit algorithms for tree search. In *Proceedings of UAI'07*, 2007.
7. Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.
8. Rémi Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.
9. S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203, 2008. To appear.
10. Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA, 2007. ACM Press.
11. L. Kocsis and C. Szepesvari. Bandit-based monte-carlo planning. In *ECML'06*, pages 282–293, 2006.
12. Liva Ralaivola, Lin Wu, and Pierre Baldi. SVM and pattern-enriched common fate graphs for the game of Go. In *Proceedings of ESANN 2005*, pages 485–490, 2005.
13. Yizao Wang and Sylvain Gelly. Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182, 2007.